C/C++プログラムの自動ベクトル化/自動並列化

日本電気株式会社 第一コンピュータソフトウェア事業部　工藤 淑裕

概　要

SX-7 システムは、一つのノード内に豊富なベクトル演算命令もつプロセッサを 32 個有しており、そのハードウェア性能を十分に引き出すためには、コンパイラの自動ベクトル化機能によりプログラムのベクトル化を行い、さらに自動並列化機能により複数のプロセッサを効率的に利用する必要がある。本文では、SX-7 システムでの C、C++プログラムの自動ベクトル化、自動並列化について、その特徴と性能向上の観点を紹介する。

1. はじめに

SX-7 は、ベクトル処理に並列処理機能を融合した「スケーラブル・パラレル・スーパーコンピュータ」です。このハードウェアのもつ高い能力をいかんなく発揮させるためには、コンパイラの自動ベクトル化/並列化機能が重要な役割を果たします。

SX の C コンパイラ、C++コンパイラは、SX-7 システムのハードウェアに密着した高度な自動ベクトル化、自動並列化機能をもつコンパイラです。その自動ベクトル化機能は、Fortran コンパイラと同等のベクトル化、拡張ベクトル化機能を持ち、さらに、C、C++言語固有のポインタ、クラスなどを利用したループのベクトル化機能が強化されています。また、自動並列化機能は、プログラムの並列性を調べ、SX-7 の実装する 32 個のプロセッサを有効に使って並列処理(共有並列処理)できるようプログラムを並列化します。

コンパイラのサポートする言語仕様は、C 言語では、ISO/IEC 9899:1990[1992](いわゆる ANSI-C)、C++言語では、ISO/IEC 14882:1998(いわゆる ISO C++、ANSI C++)であり、プログラムの移植性に優れています。

今回は、SX の C コンパイラ、C++コンパイラの自動ベクトル化、自動並列化機能をご利用していただけるよう次の点についてご紹介します。

- 自動ベクトル化機能
- 自動並列化機能
- 性能解析ツール

2. 自動ベクトル化機能

2.1. ベクトル化とは

2.1.1. ベクトル化の基本概念

for 文などを使ったループ(繰り返し処理)で行われる、規則的に並んだデータの計算に対して、コンパイラがベクトル命令を適用することを**自動ベクトル化**と呼びます。規則的に並んだデータとは、行列の要素(配列)や、ポインタを規則的にずらすことで参照、更新されるデータです。

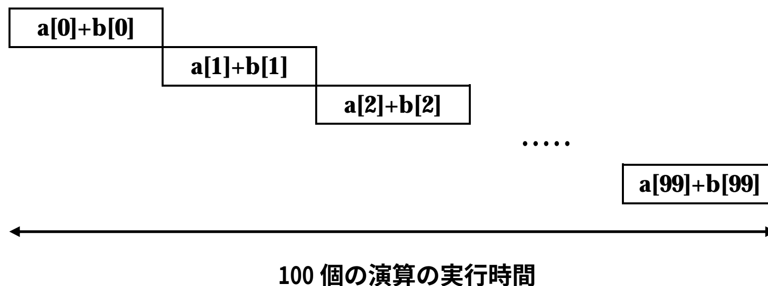
たとえば例 1のような行列(配列)を加算するループがあったとします。

例1

```
double a[100], b[100], c[100];
for (i=0; i<100; i++)
    c[i] = a[i] + b[i];
```

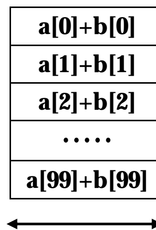
通常の演算命令では、一組のデータに対する演算処理を逐次的に実行し、計算を実行、完了します。この演算命令を、ベクトル命令と対比させるために**スカラ命令**と呼びます。

図 1 スカラ命令(スカラ加算)の実行イメージ



これに対して、**ベクトル命令**では、複数のデータに対する演算処理を一つの命令で一度に行います。

図 2 ベクトル命令(ベクトル加算)の実行イメージ



ベクトル命令での実行時間

このように、コンパイラの自動ベクトル化機能を利用して、プログラムのループをベクトル化することで、プログラムの実行時間を飛躍的に短縮することができます。

2.1.2. ベクトル化可能な条件

コンパイラは、ループ中の演算をベクトル命令で処理しても、プログラムの意図が変わらないような場合にのみ、ループをベクトル化します。

例2

```
for (i=0; i<99; i++) {
    a[i] = 2.0;         // 代入文 2-1
    b[i] = a[i+1];      // 代入文 2-2
}
```

図 3 に、このプログラムをベクトル化しない場合、ベクトル化した場合の演算の実行順序を示します。

図 3 ベクトル化した場合/しない場合の演算の実行順序

ベクトル化しない場合の実行順序

```
a[0]=2.0      // 代入文 2-1
b[0]=a[1]     // 代入文 2-2
a[1]=2.0      // 代入文 2-1
b[1]=a[2]     // 代入文 2-2
:
a[98]=2.0     // 代入文 2-1
b[98]=a[99]   // 代入文 2-2
```

ベクトル化した場合の実行順序

```
a[0]=2.0      // 代入文 2-1  ┐
a[1]=2.0      // 代入文 2-1  │ 一つのベクトル命令
:                            │
a[98]=2.0     // 代入文 2-1  ┘
b[0]=a[1]     // 代入文 2-2  ┐
b[1]=a[2]     // 代入文 2-2  │ 一つのベクトル命令
:                            │
b[98]=a[99]   // 代入文 2-2  ┘
```

例 2のループをベクトル化して実行すると、すべての要素に対して 2-1 の代入がベクトル命令で実行され、次に 2-2 の代入がやはりすべての要素に対して実行されます。このため、配列 b[0]～b[97]の値はすべて 2.0 となり、プログラムの意図と異なる結果となってしまいます。

また、n 回目の繰り返しで定義したスカラ変数を n+1 回目の繰り返しで引用する、例 3のようなプログラムも、すべての要素に対して 3-1 の代入が先に実行されるため、ベクトル化すると配列 a[0]～a[99]の値がすべてゼロとなってしまいます。

例3

```
s=0.0;
for (i=0; i<100; i++) {
    a[i] = s;           // 代入文 3-1 (スカラ変数 s の引用)
    s = b[i]+c[i];      // 代入文 3-2 (スカラ変数 s の定義)
}
```

これらのプログラムのように、ベクトル化によって配列の定義・引用関係に変化が生じてしまう場合には、コンパイラはループをベクトル化しません。

コンパイラの自動ベクトル化機能は、ソースプログラムを解析して、ベクトル命令で実行できる部分を検出するとともに、必要ならベクトル化に適合するようにプログラムを変形して、ループを自動的にベクトル化します。

2.1.3. ベクトル化による性能向上の判定

個々のベクトル命令は、演算処理に入る前にある程度の準備処理が必要となります。この準備処理に要する時間を**立ち上がり時間**と呼びます。すなわち、実際のベクトル演算に要する時間があまりにも小さいと、立ち上がり時間の影響が大きくなり、ベクトル化による高速化が期待できません。

ベクトル化した場合とベクトル化しない場合とで実行時間が等しくなるループの繰り返し数(**ループ長**)を**交差ループ長**と呼び、立ち上がり時間と交差ループ長には、図 4 に示す関係があります。

図 4 交差ループ長と立ち上がり時間

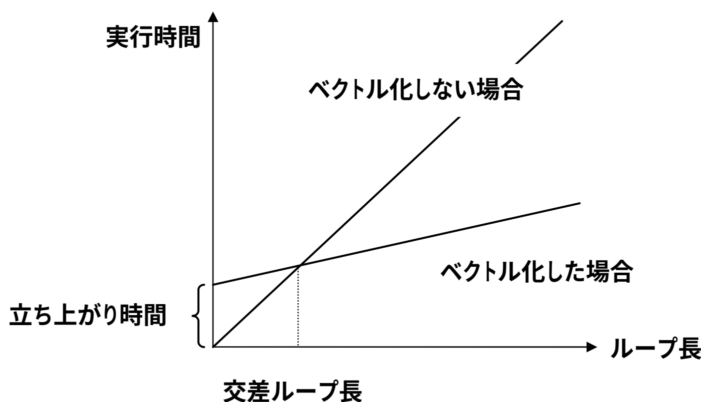


図 4 より、ループ長をできるだけ長くした方が、ベクトル化による高速化の効果が大きいことが、お分かりいただけると思います。

コンパイラの自動ベクトル化機能は、ループ長が 5 未満の場合には、ベクトル化による効果が少ないと判断し、ベクトル化を行いません。

2.2. C/C++プログラムと自動ベクトル化

コンパイラの自動ベクトル化機能は、プログラムに含まれるループを探し出し、その中で行われる計算が規則的に並んだデータの演算かを調べ、ベクトル命令を適用でき、さらに、

ベクトル化したときの性能向上が期待できるときに、ループをベクトル化します。

ここでは、C、C++コンパイラでベクトル化されるループの例をいくつかご紹介します。

(1) 行列(配列)の計算を行うループ

多次元配列の計算を行うループをベクトル化することができます。

例4

```
double a[2048][2048], b[2048][2048];
for (i=0; i<2048; i++) {
    for (j=1; j<2048; j++) {
        a[j][i] = a[j][i] * b[j][i];
    }
}
```

(2) 構造体の配列の計算を行うループ

構造体の配列の計算を行うループもベクトル化することができます。構造体のメンバが構造体であり、そのメンバを参照するような複雑な構造体であってもベクトル化することができます。

例 5は、複素数の配列 a、b、c の要素の乗算を行うループです。

例5

```
struct Complex {
    double real;
    double imag;
} a[1024], b[1024], c[1024];
for (i=0; i<1024; i++) {
    a[i].real = b[i].real*c[i].real – b[i].imag*c[i].imag;
    a[i].imag = b[i].imag*c[i].real + b[i].real*c[i].imag;
}
```

(3) ポインタを使ったループ

ポインタを規則的にずらしてアクセスするデータの計算を行うループも、「**2.5 ポインタを使用したループのベクトル化**」でご紹介する restrict 修飾子やコンパイラオプション -Orestrict を指定してベクトル化することができます。これらの修飾子、コンパイラオプションは、ポインタの指す先がループ中で参照されるどのデータとも重ならないということをコンパイラに知らせるためのものです。

例 6は、引数で受け取ったポインタ b の指す先の n 個の倍精度実数(double)データをポインタ a の指す先に代入する関数です。

例6

```
void copy(double *a, double *b, int n) {
    for (i=0; i<n; i++)
        *b++ = *a++;
}
```

引数であるポインタ a、b の値がループ中で同じにならない(重ならない)ときは、引数ポインタに restrict 修飾子を指定でき、ループをベクトル化することができます。例 7は、例 6のプログラムに対して restrict 修飾子を指定した例です。

例7

```
void copy(double * restrict a, double * restrict b, int n) {
    for (i=0; i<n; i++)
        *b++ = *a++;
}
```

(4) 構造体へのポインタを使ったループ

構造体へのポインタとそのメンバである配列を参照するような複雑なループもベクトル化することができます。

例8

```
struct Layer {
    int x[4096];
    int y[4096];
    int z[4096];
};
void merge_layer(struct Layer * restrict dest, struct Layer * restrict src) {
    for (i=0; i<4096; i++) {
        dest->x[i] += src->x[i];
        dest->y[i] += src->y[i];
        dest->z[i] += src->z[i];
    }
}
```

2.3. ベクトル化可能なループの記述

「**2.2 C/C++プログラムと自動ベクトル化**」でコンパイラによってベクトル化されるループの例を示しましたが、プログラミング時にループの記述を工夫することで自動ベクトル化機能によるベクトル化を促進し、さらにプログラムを高速化することができます。

ここでは、ベクトル化を促進するためのループの記述方法をご紹介します。

(1) できるだけ for ループを使う

コンパイラは while ループ、do-while ループもベクトル化することができますが、コンパイ

ラにとってはforループが一番ベクトル化し易いです。これはfor文で構成されるループがSXでのベクトル化に向いているためです。ループを記述するときはできるだけfor文を使ってループを記述してください。

例9

```
int i,n;
i = 0;
while (i < n) {
    // 処理
    i++;
}
```

↓ **(forループに変更)**

```
int i,n;
for (i = 0; i < n; i++) {
    // 処理
}
```

(2) インダクション変数の精度変換を行わない

ループの繰り返しの進行に伴い、一定の割合で増加、減少する変数をインダクション変数と呼びます。インダクション変数はFortranのDO変数、指標変数に相当します。Fortranプログラムの例10では、IがDO変数で、Jが指標変数です。FortranのDO文は、C/C++のfor文に似ています。

例10

```
J=N
DO I=1,N            ! このDO文は「for (i = 0; i < n; i++)」相当
    A(I)=0.0
    B(J)=A(I)*2
    J=J-1
ENDDO
```

コンパイラは、インダクション変数を基にしてループの繰り返し数や規則的に並んだデータを演算しているループかどうかを調べます。インダクション変数は、ベクトル化では非常に重要な役割を果たします。

たとえば、先ほどの例9では、変数iがインダクション変数です。次の例11では、変数i、jがインダクション変数です。ループの繰り返しの進行に伴い、iは1ずつ、jはnずつ増加します。

例11

```
int i, j;
long n;
```

```
…
for (i = 0, j = 1; i < n; i++) {    // 比較時に変数 i が long 型に型変換
    // 処理
    j = j + n;
}
```

C/C++では、longとintのように違う型の整数を演算するときには、小さいサイズの整数は自動的に大きいサイズの整数に型変換されるなど、暗黙の精度変換、算術変換が行われます。型変換が行われると、コンパイラにとってインダクション変数をみつけるのが難しくなります。このような場合、インダクション変数の型変換が行われないようにプログラミングすることで、ベクトル化を促進することができます。

インダクション変数が型変換されないようにプログラミングするには、関数、あるいはループ中で利用している整数型(long、unsigned long、int、unsigned int など)を一つの型に統一するとよいです。

先ほどの例 11では、変数 i、j は int 型(32 ビット)、n は long 型(64 ビット)です。n の値が 2 ギガより小さい値にしかならないのであれば、n の型を int 型に変更します。

例12

```
int i, j;
int n;
…
for (i = 0, j = 1; i < n; i++) {
    // 処理
    j = j + n;
}
```

(3) ループの終了判定の条件式は一つの論理式にする

ループの終了判定の条件式が複数の論理式からなるとき、ループからの飛び出しが複数できてしまいベクトル化できないことがあります。

例 13の for ループでは下線部がループの終了判定の条件式です。このループは「(インダクション変数 i が変数 n より小さい)、かつ、(p[i]の値が 0 である)」間ループを繰り返します。

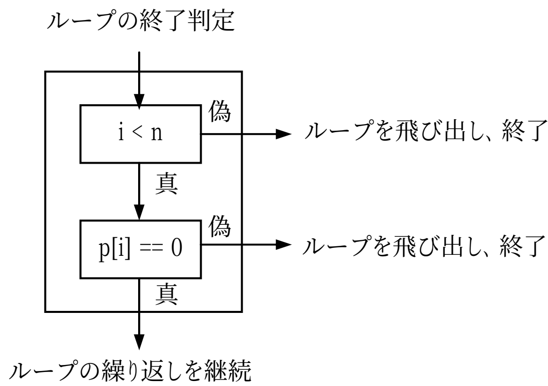
例13

```
int i, n;
double *p, *q;
…
for (i = 0; (i < n) && (p[i] == 0); i++) {
    *p++ = *q++;


}
```

ループの終了判定の流れは次のようになります。

図 5 ループの終了判定の流れ



ループの終了判定の条件式が複数の論理式からなるときには、条件式の一部をループ中に記述するようにすると、ベクトル化できるようになります。例 14は先ほどの例 13のループを書き直したものです。条件式の一部をループ中に移動しています。for 文にはインダクション変数を比較する式を残します。

例14

```
int i, n;
double *p, *q;
…
for (i = 0; i < n; i++) {
    if (p[i] != 0) break;
    *p++ = *q++;
}
```

2.4. ビット演算、バイト演算を含むループのベクトル化

SX では、4 バイト、または、8 バイトサイズのデータの演算をベクトル命令で処理することができますが、それより小さいサイズのデータはそのままではベクトル命令で処理することができません。

	データ型
4、8 バイトサイズのデータ	int、unsigned int、long、unsigned long、float、double
上記以外のデータ	char、signed char、unsigned char、short、unsigned short、long double

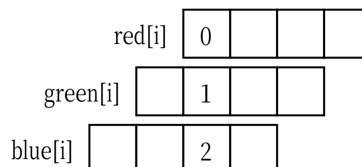
C/C++のプログラミングでよく使用される 1 バイト、2 バイトサイズのデータの演算や、4 バイトより短い長さのビット列の論理演算は、それらを int、long 型で行うようにプログラミングす

ることで、ベクトル命令で処理できるようになります。

例 15は、int の配列の要素の一部を or 演算するループです。char 型のデータで or 演算するため、このままではループはベクトル化できません。

例15

```
char *b1, *b2, *b3;
int red[1024], green[1024], blue[1024], pix[1024];
int i;
for (i = 0; i < 1024; i++) {
    b1 = &red[i], b2 = &green[i], b3 = &blue[i];
    pix[i] = b1[0] | b2[1] | b3[2];
}
```



このループは、シフト演算を使って特定の位置のバイト値を取り出してから、演算するよう変更することで、ベクトル化できるようになります。シフト演算(<<、>>)、and 演算(&)、or 演算(|)はベクトル命令で処理することができます。

例 16は例 15を変更してベクトル化できるようにした例です。

例16

```
unsigned int b1, b2, b3;
int red[1024], green[1024], blue[1024], pix[1024];
int i;
for (i = 0; i < 1024; i++) {
    b1 = (red[i] >> 24)     & 0xffU;
    b2 = (green[i] >> 16)  & 0xffU;
    b3 = (blue[i] >> 8)     & 0xffU;
    pix[i] = b1 | b2 | b3;
}
```

このように、4 バイトより小さいサイズのデータは、シフト演算、and 演算、or 演算を使って必要なビット列を取り出し、int 型のデータに代入してから演算するようにプログラミングすることで、ベクトル処理できるようになります。

2.5. ポインタを使用したループのベクトル化

この節では、C、C++言語の特徴的な言語仕様であるポインタを使用したループのベクトル化について説明します。

(1) ポインタを使ったループのベクトル化

コンパイラは、ポインタ変数を介して参照されるデータに対しても、可能な限りベクトル化を試みますが、ポインタで指される領域が別の式の演算結果であるポインタ値、別のポインタ変数で参照されるなど複雑な場合には、「**2.1.2 ベクトル化可能な条件**」で示した条件を満たすかどうかがコンパイラにとってわからないことがあります。このような場合には、コンパ

イラは常に正しい実行結果が得られるように、ループをベクトル化しません。

しかし、SX では、ポインタを使用したループであっても、restrict 修飾子やコンパイラオプション -Orestrict を指定することで、ループのベクトル化を促進することができます。

(2) 用語の定義:エリアス(Alias)

あるポインタ(p)によって指示される領域が別のポインタ(q)によっても指示されるとき、あるいは、別の変数(a)の領域の全体、一部であるとき、そのポインタはエリアス(Alias)をもつといい、ポインタ q はポインタ p のエリアス、a は p のエリアスと呼びます。

例17

```
double a;
double *p, *q;
p = &a;
q = &a;
```

ポインタ p、q、変数 a がエリアスであり、それぞれメモリ上の同じ領域を示しています。

例18

```
double array[1000];
double p;
p = array;

*(p+10)=..
....=array[10];
```

配列 array はポインタ p のエリアスであり、*(p+10)、array[10]はメモリ上の同じ領域を指しています。

C++言語で利用されるリファレンスもエリアスを定義するものです。

例19

```
double a;
double &p = a;
double q;
q = a;              // 文 20-1
q = p;              // 文 20-2
```

例では、リファレンス p は変数 a のエリアスです。文 20-1 と文 20-2 は同じメモリ領域(変数 a に割り当てられた領域)の値を q に代入していることになります。

コンパイラがポインタを含むループのベクトル化を試みるときには、ポインタやリファレンスの定義、引用関係、および、ポインタの指示先の定義、引用関係を調べ、ベクトル可能

かどうかチェックします。ポインタ、リファレンスがエリアスをもつときには、ループ内に現れるそれらのエリアスとの定義、引用関係も調べます。

(3) restrict 修飾子

restrict 修飾子は C99(1999 年に定められた C 言語の新しい言語仕様)で規格に組み込まれた修飾子で、修飾されたポインタがエリアスを持たないことをコンパイラに知らせるものです。ポインタがエリアスを持たないことがわかっているときは、そのポインタの宣言で restrict 修飾子を指定すると、ループのベクトル化を促進することができます。

エリアスを持つポインタに restrict 修飾子を指定したときには、最適化、ベクトル化の副作用により予期せぬ実行結果をもたらすことがあります。エリアスを持つポインタには指定しないよう注意してください。

- C プログラムでの利用

例20

```
double func(double *p, double *q, int n)
{
        int i;
        for (i = 0; i < n; i++)
                *p++ = *q++;
}
```

↓

```
double func(double * restrict p, double * restrict q, int n)
{
        int i;
        for (i = 0; i < n; i++)
                *p++ = *q++;
}
```

ポインタ p、q は restrict 修飾されているので、コンパイラは、p、q はエリアスを持たない、つまり、p、q の指示先(配列の一部)、変数 n とは重なりがないと判断して、for ループをベクトル化します。

- C++プログラムでの利用

SX では、restrict 修飾子を C++言語のポインタ、リファレンスでも利用できるようコンパイラを拡張しています。

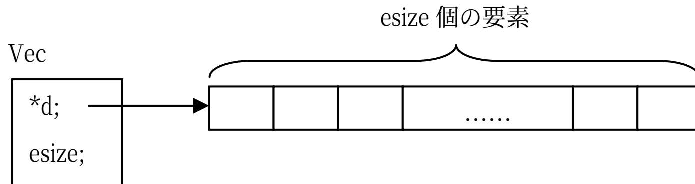
例21

```
class Vec {
        double *d;
        int esize;
```

```
public:
        Vec(int i);
        Vec & operator=(const Vec &x);
        …
};
Vec & Vec::operator=(const Vec & x) {
        for (int i=0; i<esize; i++)
                d[i]=x.d[i];
        return *this;
}
```

このクラスは可変個の要素をもつ double 型の一次元配列を定義します。esize は配列の要素数です。このクラスを利用したときのデータ構造は図 6 のようになります。

図 6 Vec クラスの構造



オーバーロードされる代入演算子(=)では for ループを使い配列のコピーを行います。引数でリファレンスを使用しているので、このままだと for ループはベクトル化されません。また、クラスのデータメンバである d、esize へのアクセスも this ポインタを介して行われるのでベクトル化できません。this ポインタを明示的に記述したときのループは次のとおりです。

```
        for (int i=0; i<this->esize; i++)
                this->d[i]=x.d[i];
```

つまり、このループで利用されているポインタ、リファレンスは、this、x、d で、これらがエリアスをもたないことがわからないとコンパイラはベクトル化を行いません。

このループの場合、this ポインタの指す領域とリファレンス x の領域が同一でないとき、すなわち、代入元、代入先の Vec クラスのオブジェクトが同一でないとき、さらに、クラスオブジェクトが異なればその Vec クラスの配列の領域(d の指す領域)が異なるようなとき、ポインタ、リファレンスはエリアスをもたないので、restrict 修飾子を指定してベクトル化を促進することができます。

次が restrict 修飾子の指定例です。

例22

```
class Vec
```

```
{
	double * restrict d;
	int esize;
public:
	Vec(int i);
	Vec & operator=(const Vec & restrict x) restrict;
	…
};

Vec & Vec::operator=(const Vec & restrict x) restrict
{
	for (int i=0; i<esize; i++)
		d[i]=x.d[i];
	return *this;
}
```

引数宣言部の後ろに指定された restrict 修飾子が this ポインタに対する restrict 修飾子の指定です。const 修飾子などと同じ方法で指定します。

(4) コンパイラオプションによる restrict 修飾子の指定

restrict 修飾子はソースプログラムに直接記述する他に、コンパイラオプションで特定の種類のポインタに自動的に付加することができます。

- オプションの形式

  -Orestrict={ all ¦ arg ¦ this ¦ keyword ¦ no }

- 説明

  -Orestrict=all と指定されたとき、コンパイラはすべてのポインタ、リファレンスが restrict 修飾されているものとみなして、ベクトル化を試みます。

  -Orestrict=arg と指定されたとき、コンパイラは引数として現れたポインタ、リファレンス、および、this ポインタが restrict 修飾されているものとみなして、ベクトル化を試みます。

  -Orestrict=this と指定されたとき、コンパイラはすべての this ポインタが restrict 修飾されているものとみなして、ベクトル化を試みます。(-Caopt、-Chopt を指定したときの既定値)

  -Orestrict=keyword と指定されたとき、コンパイラはソースプログラムで restrict 修飾子の指定されたポインタ、リファレンスのみが restrict 修飾されているとみなして、ベクトル化を試みます。(-Cvopt、-Csopt を指定したときの既定値)

  -Orestrict=no と指定されたとき、コンパイラは restrict 修飾子を無視してベクトル化を試みます。このとき、すべてのポインタ、リファレンスが restrict 修飾されていないとみなされます。プログラムのデバッグ時に指定することで、restrict 修飾子の副作用がないかを調べるときに利用します。

たとえば、例 20のプログラムは次のようにオプションを指定すると、プログラムの修正なし

で、ベクトル化できるようになります。

```
sxcc  -Orestrict=arg   example20.c
```

2.6. 拡張ベクトル化機能

自動ベクトル化機能は、ベクトル命令で実行可能な部分を自動的に検出しベクトル化しますが、そのままでベクトル化できない場合には、プログラムを変形してベクトル化したり、プログラムを変形することによってベクトル化の効果をさらに高めます。これを**拡張ベクトル化機能**と呼びます。ここでは、コンパイラの拡張ベクトル化機能のうち主なものを紹介します。

拡張ベクトル化機能を利用するときには、コンパイラオプション-Chopt または-Caopt を指定してください。

2.6.1. 文の入れ換え

「**2.1.2 ベクトル化可能な条件**」の例 2 は「ベクトル化できない」と書きましたが、実は二つの文を入れ換えるとベクトル化できることに気づかれたでしょうか？

例23 <u>**ソースプログラム**</u>

```
for (i=0; i<99; i++) {
    a[i]=2.0;       //  代入文 23-1
    b[i]=a[i+1];    //  代入文 23-2
}
```

<u>**コンパイラによる変形**</u>

```
for (i=0; i<99; i++) {
    b[i]=a[i+1];    //  代入文 23-2
    a[i]=2.0;       //  代入文 23-1
}
```

このプログラムを右のように変形すると、ベクトル化した場合でも配列の定義・引用関係に矛盾は生じないので、コンパイラは自動的に 23-1 と 23-2 の文を入れ換えてベクトル化を行います。

2.6.2. ループの入れ換え

多重ループの場合には、通常は最も内側のループをベクトル化します。しかし、ループを入れ換えることにより定義・引用関係の矛盾が解消されてベクトル化できるようになる場合や、内側のループよりも外側のループの方がループ長が長く、入れ換えた方が速いと判断した場合には、コンパイラがループを入れ換えてベクトル化を行います。

例24 構造体 a が持つ配列に依存関係がありベクトル化できない

```
for (j=0; j<a->szx; j++) {                  //  外側ループ
    for (i=0; i<a->szy; i++) {              //  内側ループ
        a->d[j][i+1] = a->d[j][i] + b->d[j][i];
    }
}
```

↓　**コンパイラによる変形のイメージ**

```
for (i=0; i<a->szy; i++) {                    // 元の内側ループ
    for (j=0; j<a->szx; j++) {                // 元の外側ループ
        a->d[j][i+1] = a->d[j][i] + b->d[j][i];
    }
}
```

依存関係がなくなりベクトル化できる。

2.6.3. 条件ベクトル化

配列の依存関係がベクトル化に適合しているかどうかコンパイル時に不明な場合や、ループ長がコンパイル時に不明で、ベクトル化した方が速いかどうかわからない場合に、ベクトル化したコードとベクトル化しないコードの両方を生成しておき、プログラムを実行するとき、そのどちらかを選択して実行するものです。

例25　k が 0 以上であるか、または、k が -100 以下ならば依存関係に矛盾が生じないのでベクトル化できる

```
for (i=0; i<100; i++)
    a[i] = a[i+k] + b[i];
```

↓　**コンパイラによる変形のイメージ**

```
if (k >= 0 || k < =-100) {
#pragma cdir nodep
    for (i=0; i<100; i++)        // ベクトル化する
        a[i] = a[i+k] + b[i];
} else {
#pragma cdir novector
    for (i=0; i<100; i++)        // ベクトル化しない
        a[i] = a[i+k] + b[i];
}
```

ポインタを使用していて依存関係が不明なときも、条件ベクトル化を行います。例 26では、a の指示先と b と c の指示先が n 以上離れていると、指示先が重なることがないのでベクトル化できます。また、a の指示先が b と c の指示先より小さいと依存関係に矛盾が生じないので、ベクトル化できます。

例26

```
for (i=0; i<n; i++)
    (*a++) = (*b++) + (*c++);
```

↓　**コンパイラによる変形のイメージ**

```
if (((a <= b) || (a - b >= n)) && ((a <= c) || (a - c >= n))) {
```

```
#pragma cdir nodep
        for (i=0; i<n; i++)
              (*a++) = (*b++) + (*c++);
} else {
#pragma cdir novector
        for (i=0; i<n; i++)
              (*a++) = (*b++) + (*c++);
}
```

2.6.4. マクロ演算の認識

次のようなパターンは、配列の定義･引用関係に矛盾があり、本来はベクトル化できませんが、コンパイラが特別なパターンであることを認識し、特別なベクトル命令を用いることで、ベクトル化を行います。

(1) 総和

例27

```
while (a<end) {
      s += *a++;
}
```

n 回目の繰り返しで定義したスカラ変数 s の値を、n+1 回目の繰り返しで引用するため、このままではベクトル化できませんが、ポインタ a の指示先の総和を求めるパターンであるとコンパイラが認識して、特殊なベクトル命令を用いることでベクトル化します。

(2) 漸化式

例28

```
for (i=0; i<n; i++) {
      a[i] = a[i-1] * b[i] + c[i];
}
```

配列 a の定義･引用関係に矛盾があるので、このままではベクトル化できませんが、特殊なベクトル命令を用いてベクトル化します。(仮に a[i-1]=a[i]*b[i]+c[i]であったとすると、定義･引用関係に矛盾はなく、そのままベクトル化できます。)

(3) 最大/最小値

例29

```
for (i=0; i<n; i++) {
      if (xmax < x[i])
            xmax = x[i];
}
```

2.6.5. ループ融合

コンパイラは同じ形状のループ構造(ループのネスト数、繰り返し回数が同じ)を一つにまとめてベクトル化します。これを**ループ融合**と呼びます。次の二つのループは形状が一致しているので、下の二重ループと同じように解釈、最適化されてベクトル化されます。

例30

```
for (j=0; j<n; j++)
    for (i=0; i<m; i++)
        a->d[j][i] = b->d[j][i] + c->d[j][i];
for (j=0; j<n; j++)
    for (i=0; i<m; i++)
        d->d[j][i] = e->d[j][i] + f->d[j][i]+s;
```

↓ コンパイラによる変形のイメージ

```
for (j=0; j<n; j++) {
    for (i=0; i<m; i++) {
        a->d[j][i] = b->d[j][i] + c->d[j][i];
        d->d[j][i] = e->d[j][i] + f->d[j][i]+s;
    }
}
```

コンパイラは、同じ形状のループ構造が連続していれば融合しますが、間に形状の異なるループ構造や、他の文があると融合できません。高速化のためには、できるだけ同じ形状のループ構造を連続させるようにしてください。

次の例では、二つのループの間に形状の異なるループ構造があるために、二つのループは融合されません。このような場合には、ループの順序を入れ換えて、同じ形状のループが連続するように書き換えてください。

例31

```
for (j=0; j<n; j++)
    for (i=0; i<m; i++)
        a->d[j][i] = b->d[j][i] + c->d[j][i];
for (i=0; i<l; i++)
    x[i] = 0.0;
for (j=0; j<n; j++)
    for (i=0; i<m; i++)
        d->d[j][i] = e->d[j][i] + f->d[j][i]+s;
```

↓ (書き換え)

```
for (j=0; j<n; j++)
    for (i=0; i<m; i++)
        a->d[j][i] = b->d[j][i] + c->d[j][i];
for (j=0; j<n; j++)
```

```
        for (i=0; i<m; i++)
            d->d[j][i] = e->d[j][i] + f->d[j][i]+s;
    for (i=0; i<l; i++)
        x[i] = 0.0;
```

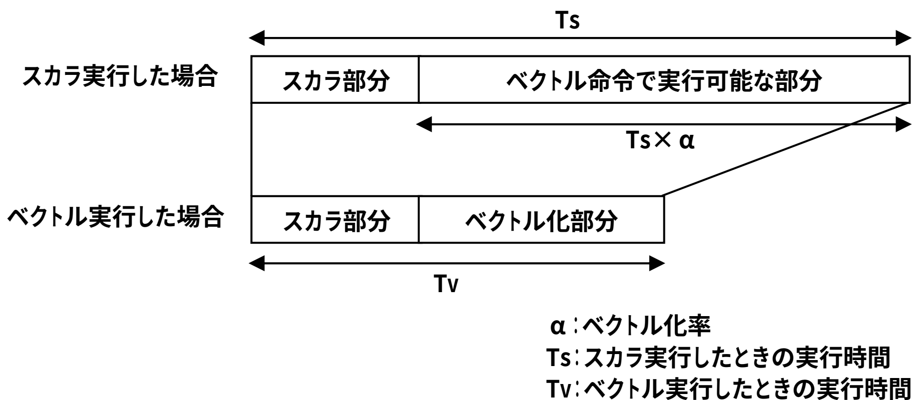
2.7. ベクトル化率向上のための手法

2.7.1. ベクトル化率

プログラムをスカラ命令だけで実行させた場合の実行時間に占める、ベクトル命令で実行可能な部分の時間の割合を**ベクトル化率**と呼びます。

エラー!

図 7 ベクトル化率



プログラムをベクトル化することにより実行性能が向上しますが、プログラムの一部だけをベクトル化しても、ベクトル化の効果はあまり期待できません。ベクトル化部分の実行時間が非常に小さくても、ベクトル化率が 50%程度では、高々2 倍の性能にしかならないことは、図 7 からもわかると思います。一般にはベクトル化率が 90～95%以上ないと、ベクトル化による大きな効果は期待できません。

すなわち、ベクトル化による高速化技法の一つは、ベクトル化率を高め、100%に近づけることにあります。

しかし、一般にベクトル化率を正確に求めることは困難であるため、SX-7 では、ベクトル化率に近い値として、プログラム特性情報(proginf 情報)に表示される**ベクトル演算率**を用いています(「4.1 proginf 情報」参照)。ベクトル演算率は、実行された命令の数をハードウェアがカウントすることで、プログラムで処理された全演算数に占める、ベクトル演算命令で処理された数の割合を求めたものです。

SX-7 では、この**ベクトル演算率**を高くすることを目標に、チューニングを行ってください。

以降では、ベクトル化率(ベクトル演算率)を向上させる手法についてご紹介します。

2.7.2. コンパイラ指示行の挿入による高速化

コンパイラは、プログラムに対して自動的に最適なベクトル命令を生成しますが、たとえば変数のもつ値のように、コンパイラがプログラムを解析してもわからない情報があると、必ずしも十分なベクトル化が行われるとは限りません。

このような場合に、コンパイラが知り得ない情報を利用者が与えることにより、ベクトル化の効果を一層促進させるものが、**コンパイラ指示行**です。

コンパイラ指示行は、

#pragma△cdir△コンパイラ指示オプション

の形式で書きます。△は一文字以上の空白を意味します。

以下に、主なコンパイラ指示オプションとその使い方について説明します。

a) vector/novector

直後のループを自動ベクトル化の対象とする(vector)か、対象としない(novector)ことを指定します。

一般に vector を指定する必要はありませんが、ベクトル長が小さく、ベクトル化しない方が効率の良いことがわかっているような場合に、novector を指定します。

例32

```
#pragma cdir novector
    for (i=0; i<m; i++)
        a[i] = (b[i] * c[i]) + (d[i] * e[i]) – (f[i] * g[i]);
```

たとえば「m は、1 または 2 にしかなり得ない」ことを利用者が知っている場合、novector を指定して、ベクトル化を抑止したほうが効率がよい

b) nodep

配列の定義・引用関係がコンパイル時に不明で、自動ベクトル化できない場合に、利用者が「定義・引用関係に矛盾がないため、ベクトル化するように」と指示するものです。

例33

```
#pragma cdir nodep
    for (i=0; i<N; i++)
        a[i] = a[i + nk];
```

nk の値が正であればベクトル化できる。条件ベクトル化で nk の値を判断するコードが出力されるが、「nk の値が常に正である」ことを利用者が知っている場合、nodep を指定することにより無条件にベクトル化される。

例34

```
#pragma cdir nodep
    for (i=0; i<N; i++)
        a[ ip[ i ] ] = a[ ip[ i ]] + b[ i ];
    }
```

ip[i]の値に重複するものが無ければベクトル化できるが、もし重複するものがある場合にはベクトル化できない。通常コンパイラにはどちらか判断できず、またこの場合には条件を判断する条件ベクトル化もできない。もし利用者が、「ip[i]の値に重複するものがない」ことを知っている場合には、nodep を指定することによってベクトル化することができる。

コンパイラ指示行では、ここで紹介した他にもいろいろなコンパイラ指示オプションが指定でき、ベクトル化を制御することができます。詳細につきましては、「C++/SX プログラミングの手引 第 3 章 コンパイラ指示行」をご参照ください。

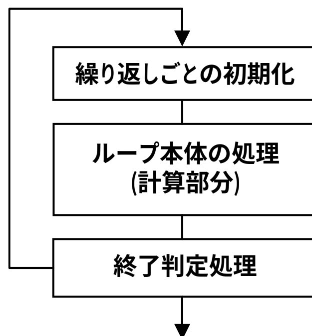
2.7.3. ループ長の短いループの高速化

この節では、ループ長(ループの繰り返し数)の短いループの高速化について説明します。

(1) ループ長とベクトル命令の効率

ループ長が極端に短いループの場合、交差ループ長は 5 なので、繰り返し数が極端に短いとベクトル命令を使わない方が高速です。また、ループ長が極端に短くない場合でも、ループ長が短いときには、ループの繰り返しごとの初期化、終了判定処理などのループの繰り返し制御の時間の占める割合が増えてしまいます。

図 8 ループの構造



(2) ループ長が極端に短いループの高速化

ループ長が極端に短い(ループ長が数回)ときは、ループをベクトル化せず、ループを展開するようにします。これによりプログラム中にループの計算部分のみを残し、ループの繰り返しごとの初期化、終了判定処理の時間を削除します。

例35

```
for (i=0; i<4; i++)
    a[i] = i;
```

↓ (ループ展開)

```
a[0] = 0.0;
a[1] = 1.0;
a[2] = 2.0;
a[3] = 3.0;
```

コンパイラは、ループの繰り返し数が 4 回以下のループを自動的にループ展開します。繰り返し数が 4 回より大きいループをループ展開したいときには、コンパイラオプションを指定するか、ループ展開のためのコンパイラ指示行を指定します。

例36 繰り返し数が 8 回のループをループ展開するためのコンパイラオプションの指定

```
% sxcc –pvctl,expand=8 a.c
```

例37 コンパイラ指示行の指定

```
#pragma cdir expand
    for (i=0; i<8; i++)
        a[i] = i;
```

ループ展開は、ループ長の短いループが最内側にあるような多重ループのベクトル化にも効果があります。コンパイラがベクトル化を試みるループは最内側ループです。最内側ループよりその外側のループのループ長が長いときは、最内側ループを展開し、その外側ループをベクトル化した方が効率がよいです。

例38

```
for (j = 0; j < N; j++)
    x[j] = y[j] * z[j];
    for (i=0; i<5; i++)      // 最内側ループ(ベクトル化対象)
        a[j][i] = i;
```

↓ (ループ展開)

```
for (j = 0; j < N; j++) {       // 最内側ループ(ベクトル化対象)
    x[j] = y[j] * z[j];
    a[j][0] = 0.0;
    a[j][1] = 1.0;
    a[j][2] = 2.0;
    a[j][3] = 3.0;
    a[j][4] = 4.0;
}
```

(3)　ループ長がベクトルレジスタ長(256 回)以下のループ

SX システムのベクトル演算命令では、一つの命令で一度に最大 256 要素のデータを処理することができます。たとえば、例 39のような 512 個の配列要素を処理するループ長が 512 回のループでは、計算のためのベクトル命令列が 2 回繰り返されます。これは、一回のベクトル命令列で 256 個の要素を計算できるからです。

例39　ループ長が 512 回のループ

```
for (i=0; i<512; i++)
    c[i] = a[i] + b[i];
```

↓ **(計算のために実行されるベクトル命令)**

```
VR1 ← a              // 配列 a から 256 個の要素をロード   ┐
VR2 ← b              // 配列 b から 256 個の要素をロード   │ 1 回目の繰り返し
VR3 ← VR1 + VR2      // ロードした 256 個の要素を加算      │
c ← VR3              // 256 個の計算結果を配列 c にストア  ┘
VR1 ← a              // 配列 a から 256 個の要素をロード   ┐
VR2 ← b              // 配列 b から 256 個の要素をロード   │ 2 回目の繰り返し
VR3 ← VR1 + VR2      // ロードした 256 個の要素を加算      │
c ← VR3              // 256 個の計算結果を配列 c にストア  ┘
```

仮に例 39のループのループ長が 256 回だとしたらどうでしょうか? そのときはループの 2 回目の繰り返しは不要です。また、ループの繰り返し制御のための処理(図 8 の終了判定処理)も不要になります。

ループ長が 256 回以下のループをショートループと呼び、コンパイラはショートループに対して終了判定処理のための命令を作成せず、ループを高速に実行できるようにします。

ループ長が実行時にならないとわからないような例 40の場合は、コンパイラ指示行 shortloop を記述することでコンパイラにショートループの命令列を作成するよう指示することができ、高速化できます。

例40

```
void func(int n) {
    …
#pragma cdir shortloop
    for (i=0; i<n; i++)
        a[i] = b[i] + c[i];
}
```

2.7.4. 等値演算子(!=、==)を使ったループ

SX では、等値演算子(!=、==)を使ったループもベクトル化することができます。ベクトル化するときには、コンパイラオプション -pvctl,loop_eq を指定します。次の例 41のループはコ

ンパイラオプション-pvctl,loop_eq を指定するとベクトル化できます。

例41

```
double sum(double * restrict first, double * restrict last) {
    double s = 0.0;
    while (first != last) {
        s += *first;
        first++;
    }
    return s;
}
```

ただし、SX のプログラミング手法としては、ループの終了判定では等値演算子をできるだけ使わないようにすることをお勧めします。等値演算子を使わない方が、コンパイラオプションの指定なしでベクトル化できるとともに、ループが無限に繰り返されてしまうようなプログラムミスも防ぐこともできます。たとえば、例 41で while 文の条件式が偽にならないとき、すなわち、first と last の値が一致しないとき、while ループは無限に回り続けるループとなります。

例 42は先の例 41を修正した例です。ループは-pvctl,loop_eq を指定しなくてもベクトル化されます。また、引数で渡されるポインタの first、または、last の値が誤っており、一致することがなかったとしてもループが無限に回るような問題は発生しなくなります。

例42

```
double sum(double * restrict first, double * restrict last) {
    double s = 0.0;
    while (first < last) {
        s += *first;
        first++;
    }
    return s;
}
```

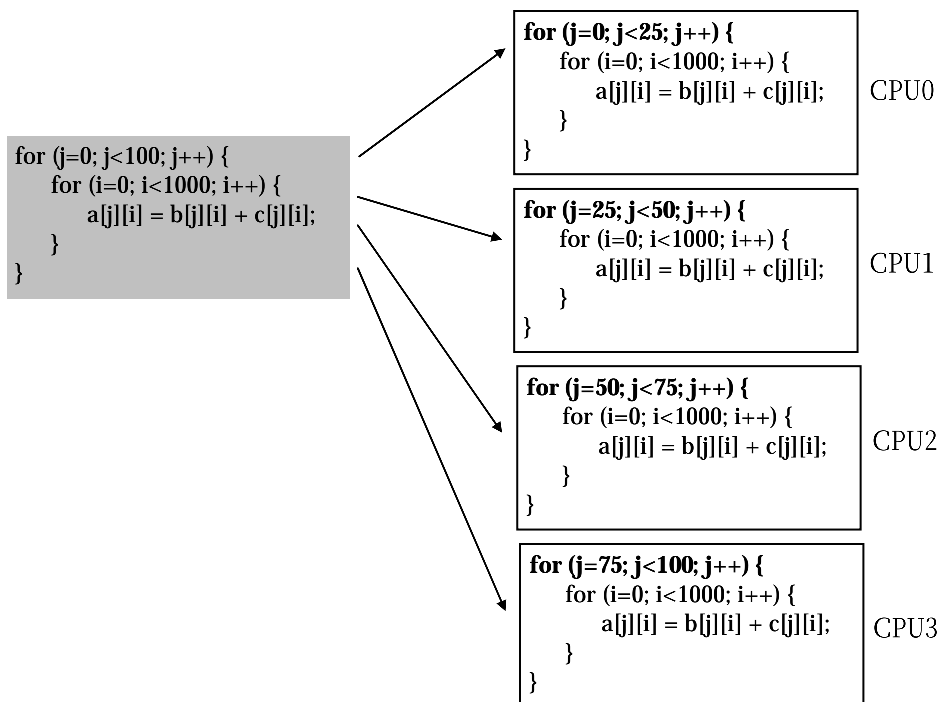
3. 自動並列化

3.1. 並列処理とは

並列処理とは、1 つの仕事をいくつかの小さな仕事に分割し、それを複数のタスク（CPU）で並列に実行することです。コンパイラが備えている自動並列処理機能とは、「コンパイラがプログラムを解析して、並列に実行可能なループや文の集まりを抽出し、ループの繰り返しや文の集まりを複数のタスクに自動的に割り当てて実行時間を短縮する機能」です。

コンパイラが、ループを 4 つのタスクに分割して実行するイメージは、図 9 ようになります。この例では、外側ループの 100 回の繰り返しを 4 つに分割して、各 CPU 上で各々を並列に実行します。

図 9 並列化の実行イメージ



この例の配列 a は分割されて、各タスク毎に値（a[0][i]～a[24][i]、 a[25][i]～a[49][i]、a[50][i]～a[74][i]、 a[75][i]～a[99][i]）が計算され、定義されるので、a は各タスクから共通に参照できるグローバルな領域に割り当てられます。このような各タスクから共通に参照で

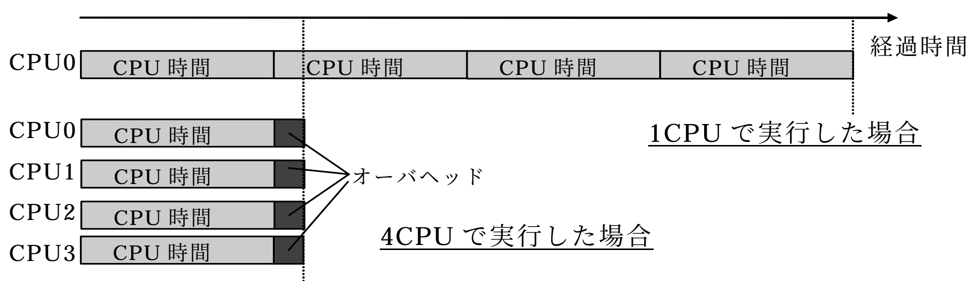
きるデータをタスク間共有データと呼びます。これに対して、並列実行される各タスクから、非同期に定義/参照を行うと、結果が不正になってしまうようなデータ（上例ではi）は、各タスク毎にローカルな領域（スタック）に割り当てられ、タスク固有データと呼びます。自動並列化機能は、このようなデータの割当ても適切に行います。

この自動並列化機能は、コンパイラオプション"-Pauto"を指定するだけで利用できます。

```
sxc++ -Pauto program.C
```

ここで、"実行時間の短縮"には注意が必要です。並列処理は、1つの仕事を分割して、並列に実行を行うわけですから、CPU時間が削減されるわけではなく、経過時間が短縮されることになります。また、仕事を各タスクで並列実行させるための処理（オーバーヘッド）も必要となり、CPU時間は、かえって増加することになります。たとえば、CPU時間と経過時間の関係は、図10のようになります。

図 10 並列化の実行時間



3.2. 並列処理とベクトル処理

ここで、"ベクトル化による実行時間の短縮"と"並列化による実行時間の短縮"との相違を明確にしたいと思います。ベクトル処理とは、規則的に並んだ複数個の配列データを一度に演算する高速なベクトル命令を使って処理を行うことであり、この場合、CPU時間が短縮され、同時に経過時間も短縮されます。これに対して、並列処理では、先に述べた通り、合計のCPU時間は並列化のオーバヘッドにより、単一CPUで実行した時よりも増加することになりますが、経過時間を短縮することによって高速化を図ります。したがって、ベクトル化の場合は、単一CPUでの実行ですが、上手にベクトル化できれば、スカラで実行した時よりも、一般的に10倍以上の性能向上が期待できます。並列化の場合に期待できる性能向上の効果は、最大で使用可能なCPUの個数倍となります。

これらのことより、基本的には、ベクトル化と並列化を組み合わせて利用し、多重ループ

の内側ループについてはベクトル化を行い、外側ループを並列化することが、高速化を図る最善の方法となります。

また、並列処理した場合には、オーバヘッド時間が加わりますので、並列に実行される仕事量（粒度と呼びます）が十分に大きくなければ、並列化の効果は期待できません。当然ですが、並列処理のオーバヘッド時間よりも並列実行される部分の実行時間の方が小さければ、並列化したことにより、実行時間（経過時間）がかえって多くなってしまうことになります。

ベクトル化できるプログラムは、ベクトル化すればほとんどすべての場合に性能向上が図れますが、並列化できるプログラムは、並列化したからといって必ずしも性能が向上するとは限らないことになります。すなわち、どんなプログラムでも自動並列化すれば性能が向上するということではないことに注意して下さい。

以降で、効果的な並列化の方法について、説明していきます。

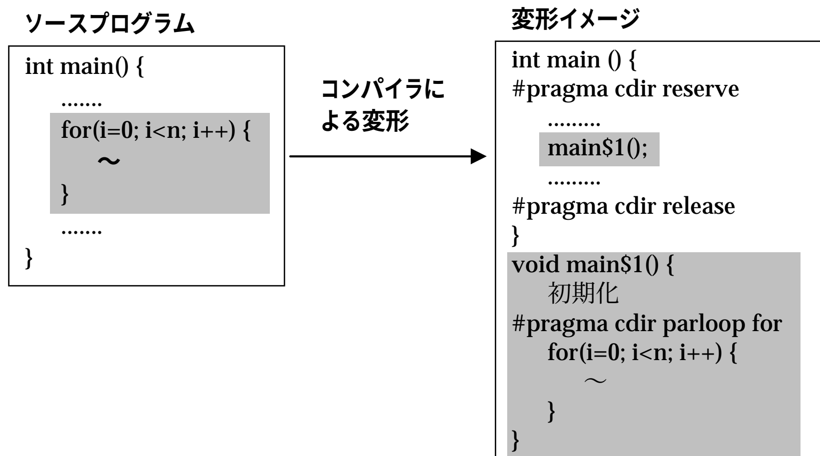
3.3. 自動並列化

コンパイラの自動並列化機能を使用すれば、容易にプログラムを並列化することができます。自動並列化機能は、プログラムを解析し、並列化した場合の効果も調べて、並列化を行います。たとえば、並列化すれば十分性能が向上するだけの粒度(ループの繰り返し数、ループの演算量)をもっているか、ループの繰り返しを並列実行しても結果不正になるような文を含んではいないかなどを調査し、可能な場合は、並列化できるようにループやデータの定義を書き直して並列化を行います。

自動並列化は、内部的に次の例のようなイメージで行われます。

コンパイラが並列可能なループを検索し、そのループを新たに関数として切り出し、マイクロタスク機能を使って並列化します。並列実行されるループを関数として切り出すことによって、並列化効率を最大限に引き出すことが可能となります。

例43



上記例の#pragma cdir で始まる行はコンパイラによって挿入された指示行で、reserve はタスクの確保、release はタスクの解放、parloop はループを並列実行することを指示します。また切り出された関数名には、元の関数名に$1、$2、…とサフィックスが付きます。

3.4. 自動並列化方法

並列実行される場合は、先にも述べたとおり、粒度が十分に大きくなければその効果が期待できません。また、SX-7 はベクトルマシンであるため、ベクトル化が性能向上には欠かせない要因となります。これらのことを考慮し、自動並列化機能は、基本的には、多重ループの内側ループをベクトル化し、外側ループを並列化します。内側ループがベクトル化できない場合は、スカラコードのまま外側ループが並列化されます。

例44

```
void fun(double ***a, int *h)
{
    int i, j, k;
    for (i=0; i<100; i++) {                          // 並列化
        for (j=0; j<100; j++) {
            for (k=0; k<999; k++) {                  // ベクトル化
                a[i][j][k] = (a[i][j][k+1] + a[i][j][k]) * 0.5;
            }
        }
        h[i] = h[i] + 1;
    }
}
```

さらに、コンパイラは、並列化の効果を高めるため、可能であればループ変形などの最適化を行い、並列化を促進します。これらの並列化の工夫について、いくつか例を紹介します。

- 一重ループの場合

基本的には、ベクトル化を行いますが、ループのコストが大きい、つまり粒度が大きい場合は、ループを分割して、ベクトル化＋並列化を行います。ベクトル化できない場合は、並列化だけが行われます。

例45

```
for (i=0; i<100; i++) {            // ベクトル化
    a[i] = b[i] + c[i];
```

```
}

for (i=0; i<10000; i++) {           // ベクトル化+並列化
    a[i] = b[i] + c[i];
}
```

- ループ融合が可能なループの場合

ループ融合やループアンロールなどのループの最適化を行った後に、並列化を行います。

例46

```
void fun(
    double **a,
    double **b,
    double **c)
{
    for (j=0; j<4; j++)
        for (i=0; i<10000; i++)
            a[j][i] = b[j][i] * b[j][i];
    for (j=0; j<4; j++)
        for (i=0; i<10000; i++)
            b[j][i] = c[j][i] - a[j][i];
}
```

コンパイラによる変形イメージ →

```
for (i=0; i<10000; i++) {      // 並列化
    a[0][j] = b[0][j] * b[0][j];
    a[1][j] = b[1][j] * b[1][j];
    a[2][j] = b[2][j] * b[2][j];
    a[3][j] = b[3][j] * b[3][j];
    b[0][j] = c[0][j] - a[0][j];
    b[1][j] = c[1][j] - a[1][j];
    b[2][j] = c[2][j] - a[2][j];
    b[3][j] = c[3][j] - a[3][j];
}
```

- 条件並列化

ループの粒度(繰り返し数、演算量)、あるいは依存関係が不明で、並列化の効果がコンパイル時に判断できない場合、実行時に粒度や依存関係を調べて並列コードを実行するかどうかを選択できるように条件並列化を行います。

次の例では、nx*ny>n によって、粒度が並列化するのに十分かどうかを調べています。これは、ループの繰り返し数が十分に大きいかどうかを実行時にチェックしているわけですが、このとき、コンパイラは単純に繰り返し数だけではなく、ループ中の各演算（加算や乗算など）に対して重み付けを行い、ループの演算コストから並列化の効果が十分に期待できる値（n）を計算して、条件並列化を行います。

また、ループ中のデータに依存関係ある場合は並列化すると結果不正になりますが、この例では、配列 y を定義している y[ic+i]と y[id+i]の ic と id の関係を実行時に調べること（id-ic==0 || abs(id-ic)>=nx）によって、このループの実行中に、配列 y の同じ領域にデータを書き込まない場合のみ並列化されるようにしています。

例47

```
for (i=0; i<nx; i++) {
    aa = a;
    bb = b;
    for (j=0; j<ny; j++) {
        aa = aa + x[i+1] * g[j-1];
        bb = bb + x[i-1] * g[j-1];
    }
    y[ic+i] = -aa;
    y[id+i] = -bb;
}
```

コンパイラによる
変形イメージ
→

```
if (nx*ny > n
        && (id-ic == 0 || abs(id-ic) >= nx) {
    並列コード
} else {
    非並列コード
}
```

3.5. 並列化の阻害要因

先にも述べた通り、ループの繰り返し間にデータの依存関係がある場合は、並列化はできません。並列化を妨げる要因をいくつか紹介します。

(1) 添え字に重なりがある場合

例48

```
for (i=0; i<n; i++) {
    a[i] = b[i+1];
    b[i] = c[i];
}
```

この例は、ループの繰り返しにまたがってデータの依存関係があるために並列化ができない例です。ただし、ベクトル化は可能です。依存関係によるベクトル化可/並列化不可の理由をもう少し具体的に説明します。

ベクトル化の場合は、ループの繰り返しの実行順序は保証されますが、並列化の場合は、ループの実行順序は保証されません。上記例においてループの繰り返しと配列 b の定義/参照関係に着目すると以下のようになります。この例では、ベクトル化の場合は、参照と定義の順番は保証され、たとえば、b[2]の値は必ず参照してから定義されることになります。

ループの繰り返し	参照	定義
1	b[1]	b[0]
2	b[2]	b[1]
3	b[3]	b[2]
4	b[4]	b[3]
・・・・	・・・・	・・・・

すなわち、ループの繰り返しにまたがっての定義/参照関係は、プログラム通りの正しい関係が保持されることになります。次に並列化の場合ですが、簡単のために、ルー

プの繰り返し 2 回毎に並列化する場合を例に考えてみます。

ループの繰り返し	参照	定義	
1	b[1]	b[0]	タスク 1 で実行
2	b[2]	b[1]	
3	b[3]	b[2]	タスク 2 で実行
4	b[4]	b[3]	
….	….	….	

この場合は、タスク 1、タスク 2、… が並列に実行されることになるため、b[2]の値がタスク 1 で参照されるタイミングとタスク 2 で定義されるタイミングの順序は保証できません。すなわち、先にタスク 2 で定義された値（b[2]の値）をタスク 1 で参照して、a[1]に代入してしまう可能性があるわけです。

(2) 定義と引用が閉じていない場合

例49

```
for (i=0; i<n; i++) {
    c[i] = t;
    t = b[i];
}
```

ループ中で、変数 t を引用してから、定義しているため、ループの繰り返しを並列実行すると、結果不正となります。文の意味は変わってしまいますが、次のように変数 t を定義してから、引用していれば、並列化が可能です。

```
for (i=0; i<n; i++) {
    t = c[i];
    …
    b[i] = t;
}
```

(3) if 文下にループ制御変数以外の指標変数がある場合

例50

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        if (a[j][i] >= del) {
            ii = ii + 1;
            ic[j][ii] = ii;
        }
    }
    for (i=0; i<ii; i++) {
        b[j][i] = ic[j][i] + sin(c[ii,j]);
    }
}
```

ii の更新 (if 文 then 節の実行) が a[j][i]の値に左右され、ii は外側ループの繰り返しにおいても加算されていきますが、外側ループで並列化された場合、各タスクで並列実行されるループ毎に ii が加算されてしまい、ii の値が正しく計算されなくなり、b の結果が不正となります。

(4) ループからの飛び出しがある場合

例51

```
for (j=0; j<n; j++) {
    for (i=0; i<n; i++) {
        a[j][i] = b[j][i] * b[j][i];
        if (a[j][i] >= del) break;        // ループからの飛び出し
        if (c[j][i] >= 0)
            b[j][i] = c[j][i] - a[j][i];
        else
            b[j][i] = c[j][i] + a[j][i];
    }
}
```

並列実行されると、ループの繰り返しの実行順序が保証されないため、ループから飛び出すタイミングがプログラム通りにならない場合があり、配列 b の値が不正になる可能性があります。

3.6. 指示行とプログラムの書き換えによる並列化促進

並列化の阻害要因は、上記 3.5 以外にもありますが、比較的発生し易い状況は、依存関係によって並列化が妨げられる場合や、各タスクで同じデータに書き込みを行ってしまうことによって結果不正を引き起こしてしまうような場合です。これらは、プログラミング上の工夫によって、回避することが可能な場合も多々あります。また、ベクトル化の場合と同じように、プログラマには、依存関係がないことがわかっており、それをコンパイラに教えてやることによって並列化が可能になる場合もあります。このような状況に対応できるように、コンパイラには、並列化のための指示行が用意されています。本節では、指示行の紹介と利用法、およびプログラミング上の工夫の例を紹介します。

(1) 並列化指示行

並列化指示行は、

#pragma△cdir△並列化指示オプション

の形式で書きます。△は一文字以上の空白を意味します。

並列化指示行の主な並列化指示オプションとその利用法は、次のとおりです。

a) concur/noconcur

直後のループを自動並列化の対象とする/しないを指定します。

たとえば、並列化するとかえって性能が劣化するループをプログラムが含んでいる場合に、そのループの先頭に noconcur を指定します。

b) inner/noinner

最内側ループあるいは一重ループを自動並列化の対象とする/しないを指定します。

最内側ループは、既定値では自動並列化の対象とはならないため、最内側ループを並列化できると効果がある場合に、inner を指定します。また、一重ループの場合も、並列化効果がコンパイル時に不明な場合は、並列化の対象とはなりませんが、inner を指定することによって並列化をすることが可能となります。

例52

```
#pragma cdir inner
    for (i=0; i<n; i++) {
        a[i] = b[i]*b[i] + c[i]*c[i];
    }
```

コンパイラによる
変形イメージ →

```
if (n > 250) {
    ベクトル+並列コード
} else {
    ベクトルコード
}
```

この例では、コンパイラは、ループ中のコストを計算して、条件並列化を行っています。

c) nosync

ループ中の配列要素に重なりがないことを指定します。

次の例で、k1 と k2 の値がコンパイル時に不明な場合は、k1=k2 であった場合に結果不正になる可能性があるため、このループを自動並列化することはできません。k1 と k2 の値が同じでないことがわかっている場合は、nosync を指定することによって並列化が可能となります。

例53

```
#pragma cdir nosync
    for (j=0; j<ny; j++) {              // 並列化
        for (i=0; i<nx; i++) {
            a[j+1][k1][i] = a[j][k2][i] + b[i];
        }
    }
```

(2) プログラミング上の工夫による並列化促進

- ループの入れ換えによる並列化促進

次のループは外側ループで並列化すると、a[j][i]と a[j-1][i]の間に依存関係があり、実行結果が不正となります。この場合は、ループを入れ換えることによって並列化が可

能となります。

例54

```
for (j=1; j<m; j++) {
    for (i=0; i<n; i++) {
        a[j][i] = a[j-1][i] * b[j][i];
    }
}
```

→

```
for (i=0; i<n; i++) {
    for (j=1; j<m; j++) {
        a[j][i] = a[j-1][i] * b[j][i];
    }
}
```

4. 性能解析のためのツール

4.1. proginf 情報

これまで述べてきたベクトル演算率を始め、プログラムの実行性能を調べる最も簡単で有効な情報がプログラム特性情報(proginf 情報)です。proginf 情報は、プログラムの実行時に、

setenv△C_PROGINF△YES または、**setenv△C_PROGINF△DETAIL**

を指定することで、表示されます(△は空白一文字を意味します)。東北大学では既定値として DETAIL が設定されています。

図 11 は、setenv C_PROGINF DETAIL で採取した proginf 情報の例ですが、ベクトル演算率(図 11 の②)、ベクトル長(図 11 の①)ともに非常に大きく、とても効率よくベクトル化されたプログラムでの例を示しています。

もしベクトル演算率が低ければ、コンパイラや後述の簡易性能解析機能の出力情報をもとに、ベクトル化できていないループ構造を抽出し、ベクトル化できるように変形したり、指示行を挿入したりするなどして、ベクトル演算率の向上を図ります。

次に、平均ベクトル長が短い場合には、二重以上のループ構造を抽出し、ループ展開やループ入れ換えを行い、ループ長の一番長いループがベクトル化されるようにすることで、ベクトル長を拡大します。また、ループ長が極端に短いループ構造がベクトル化されている場合には、novector 指示行を挿入して、ベクトル化を行わないことも検討します。

図 11 proginf 情報

```
      ******    プログラム  情報      ******
      経過時間（秒）              :          6.774000
      ユーザ時間（秒）            :          4.911325
      システム時間（秒）          :          0.230647
      ベクトル命令実行時間（秒）  :          4.846089
      全命令実行数                :         380150140.
      ベクトル命令実行数          :         205713526.
      ベクトル命令実行要素数      :       52468073111.
      浮動小数点データ実行要素数  :       14758745137.
      MOPS 値                     :      10718.595612
      MFLOPS 値                   :       3005.043295
   ①→平均ベクトル長              :        255.054075
   ②→ベクトル演算率（%）         :         99.668639
      メモリ使用量（MB）          :       1820.031250
      MIPS 値                     :         77.402761
      命令キャッシュミス（秒）    :          0.014333
      オペランドキャッシュミス（秒）:        0.010467
      バンクコンフリクト時間（秒）:          0.244283
```

図 12 は、4 並列で実行した時の proginf 情報です。並列処理プログラムの proginf 情報には、並列処理プログラムの性能解析のための情報(*)が表示されます。たとえば 1 台以上で実行した時間が、2 台以上で実行した時間より極端に小さいとき、プログラムの並列化部分が少ないです。このようなときはプログラムの並列化を促進してください。

図 12 並列処理時の proginf 情報

```
    ******    プログラム 情報    ******
経過時間（秒）                :        1.625086
ユーザ時間（秒）              :        4.123287
システム時間（秒）            :        0.051459
ベクトル命令実行時間（秒）    :        3.782491
全命令実行数                  :      282719454.
ベクトル命令実行数            :      129887663.
ベクトル命令実行要素数        :    33130449430.
浮動小数点データ実行要素数    :    14758810705.
MOPS 値                       :     8072.026328
MFLOPS 値                     :     3579.379923
MOPS 値    (実行時間換算)     :    31341.015790   (*)
MFLOPS 値（実行時間換算）     :    13897.551635   (*)
平均ベクトル長                :      255.070025
ベクトル演算率（%）           :       99.540815
メモリ使用量（MB）            :     1920.000000
最大同時実行可能プロセッサ数  :              4.   (*)
   1台以上で実行した時間（秒）:        1.061972   (*)
   2台以上で実行した時間（秒）:        1.058297   (*)
   3台以上で実行した時間（秒）:        1.057302   (*)
   4台以上で実行した時間（秒）:        0.945636   (*)
イベントビジー回数            :              0.   (*)
イベント待ち時間（秒）        :        0.000000   (*)
ロックビジー回数              :              0.   (*)
ロック待ち時間（秒）          :        0.000000   (*)
バリアビジー回数              :              0.   (*)
バリア待ち時間（秒）          :        0.000000   (*)
MIPS 値                       :       68.566523
MIPS 値（実行時間換算）       :      266.221194   (*)
命令キャッシュミス（秒）      :        0.003629
オペランドキャッシュミス（秒）:        0.006416
バンクコンフリクト時間（秒）  :        0.251331
```

4.2. 簡易性能解析機能(ftrace)

proginf 情報では、プログラム全体のベクトル化率(ベクトル演算率)等を知ることができますが、関数単位の性能を知ることはできません。しかし、簡易性能解析機能(ftrace 機能)を使用すれば、関数単位の性能情報を表示することができ、この情報を元にプログラム中の特定の関数の性能上の問題点を調べることができます。

簡易性能解析機能の使用例を以下に示します。

- C プログラムのとき

% sxcc -ftrace test.c	…	**–ftrace オプションを指定**
% a.out	…	**実行ファイルを実行**
% ftrace++	…	**ftrace++コマンドを実行**

- C++プログラムのとき

% sxc++ -ftrace,demangled test.cpp	…	**–ftrace,demangled オプションを指定**
% a.out	…	**実行ファイルを実行**
% ftrace++	…	**ftrace++コマンドを実行**

図 13 は、並列処理プログラムを 4 並列で実行した時の C++プログラムの性能情報を表示したものです。-microN (N:1～4)という表示は、タスクごとの情報です。

なお、ftrace++コマンドを実行する代わりに、実行ファイルの実行時に

setenv△C_FTRACE△YES

を指定することで(△は空白一文字を意味します)、プログラムの実行が終了したときに、性能情報が自動的に表示されます(東北大学では既定値として YES が設定されています)。

図 13 簡易性能情報の出力例

```
*--------------------------*
  FLOW TRACE ANALYSIS LIST
*--------------------------*
Execution : Tue Oct 15 19:17:28 2002
Total CPU : 0:00'31"942

関数名       Call回数    実行時間                                 ベクトル 平均
                                                                   演算率   ベクトル長
PROG.UNIT  FREQUENCY  EXCLUSIVE        AVER.TIME    MOPS MFLOPS V.OP  AVER.    VECTOR ……  BANK
                      TIME[sec](  % )    [msec]                  RATIO V.LEN    TIME   ……  CONF

Compute()          1    24.094( 75.4) 24094.325   268.9  123.8 89.70  25.6   18.966 ……  6.1525
compute$1        512     4.815( 15.1)     9.403  1350.0  672.5 99.63 503.3    3.807 ……  1.8695
 -micro1         128     1.531(  4.8)    11.958  1073.2  534.5 99.61 503.3    0.938 ……  0.4645
 -micro2         128     1.086(  3.4)     8.485  1504.8  749.8 99.65 503.3    0.961 ……  0.4743
 -micro3         128     1.104(  3.5)     8.622  1480.7  738.0 99.68 503.3    0.991 ……  0.4839
 -micro4         128     1.094(  3.4)     8.549  1451.5  722.9 99.60 503.3    0.917 ……  0.4467
fft_z(Complex *, int)
                 128     1.569(  4.9)    12.257   391.9  197.5 98.58  27.1    1.568 ……  0.5035
fft_z$1          512     1.352(  4.2)     2.640  1859.8  897.0 99.67 503.5    1.194 ……  0.3935
 -micro1         128     0.341(  1.1)     2.663  1602.8  772.1 99.55 503.5    0.264 ……  0.0868
 -micro2         128     0.336(  1.1)     2.628  1937.9  935.1 99.72 503.5    0.315 ……  0.1036
 -micro3         128     0.337(  1.1)     2.634  1951.6  941.7 99.72 503.5    0.316 ……  0.1035
 -micro4         128     0.337(  1.1)     2.634  1949.8  940.5 99.68 503.5    0.299 ……  0.0995
bc_z(Complex *, Complex *, int)
                2048     0.031(  0.1)     0.015   494.0  182.7 91.68  26.5    0.022 ……  0.0032
                            :
------------------------------------------------------------------------------------------------
total           4121    31.942(100.0)     7.751   506.0  242.9 95.57  57.6   25.579 ……  8.9232
```

より詳しい使い方については、「C++/SX プログラミングの手引」の「8.3 簡易性能解析機能」をご参照ください。

5. おわりに

以上、C、C++プログラムの自動ベクトル化と自動並列化についてご紹介させていただきました。今回は特に、スーパーコンピュータを初めて使われる方にも分りやすくご説明したつもりです。しかしすべてをご紹介することはできませんでしたので、さらに詳細につきましては、「C++/SX プログラミングの手引」を参照してくださるようお願い致します。

皆様が SX-7と C、C++コンパイラを使っていただく上で、本稿が、多少なりともお役に立てれば幸いです。

参考文献

C++/SX プログラミングの手引(G1AF28)